Elementi di programmazione

La documentazione dei programmi

S.Fumich

1. MOTIVAZIONI E ASSERZIONI.

1.0 Un "programma" e' la descrizione di un processo di calcolo o, piu' in generale, di elaborazione che deve essere comprensibile sia all'uomo sia a un elaboratore.

Alla macchina basta un linguaggio di tipo "procedurale", cioe' un insieme di comandi o istruzioni di tipo generalmente imperativo, traducibili da un compilatore in sequenze di comandi piu' semplici, direttamente interpretabili dall'hardware.

Questo tipo di descrizione del processo non e' certamente sufficiente all'uomo.
Per la nostra comprensione di un programma, i comandi devono essere integrati da informazioni di carattere piu' riassuntivo, che descrivano che cosa verra' fatto e perche', cioe' lo scopo di una sequenza di comandi, e le sue conseguenze, cioe' lo "stato" raggiunto dopo l'esecuzione della sequenza.

Pertanto ogni programma sara' costituito da una parte procedurale ("testo procedurale"), costituita di comandi imperativi comprensibili ed eseguibili dalla macchina, e da una parte non procedurale ("testo esplicativo"), ignorata dal compilatore del linguaggio, ma comprensibile all'uomo.
Il testo esplicativo verra' costruito passo passo assieme al testo procedurale e assolutamente non dopo.
Per la costruzione del testo esplicativo si useranno due costrutti linguistici fondamentali: le "motivazioni" e le "asserzioni".

1.1 "Motivazione" e' la descrizione dello scopo o del significato di un certo costrutto.

Converremo di esprimere la motivazione di un costrutto con una frase italiana racchiusa fra parentesi quadre, anteposta al costrutto a cui si riferisce e, opzionalmente, seguita dal simbolo ":", che potra' leggersi "cioe'".

In generale avremo tre tipi di motivazioni:
- motivazioni di segmenti, come nel seguente esempio:

```
BEGIN [STAMPA L'IVA DI IMPORTO:]
      TEMP = IMPORTO * 18
      TEMP = TEMP/100
      STAMPA TEMP
END
```

- motivazioni di condizioni, come nell'esempio:

```
IF [DISCRIMINANTE NULLO:] B ** 2 - 4 * A * C = 0
   THEN ......
```

- motivazioni di espressioni, esempio:

```
TEST [DISCRIMINANTE NULLO:] B ** 2 - 4 * A * C
     CASE .....
```

1.2 "Asserzione" e' una proposizione inserita nel testo di un programma, che specifica lo stato in cui si trovano le variabili di programma tutte le volte che il controllo passa per quel punto.

Converremo, per distinguere le asserzioni dalle motivazioni, di racchiudere le asserzioni fra parentesi graffe. (NB. in questo testo si useranno i segni "⊣" e "⊢" per rappresentare le parentesi graffe.)

Esempio:

```
BEGIN [STAMPA L'IVA DI IMPORTO:]
      TEMP = IMPORTO * 18
      TEMP = TEMP/100
      STAMPA TEMP
END ⊣TEMP = IVA DI IMPORTO⊢
```

1.3 Consideriamo il seguente testo:

```
..............
istruzione-1
⊣X = 0⊢
istruzione-2
..............
```

Osserviamo che l'asserzione ⊣X = 0⊢ esprime la condizione di uscita da istruzione-1 e la condizione di ingresso in istruzione-2: l'asserzione ⊣X = 0⊢ e' dunque una "condizione di interfaccia" fra le due istruzioni.

1.4 Per definizione, una asserzione deve essere verificata tutte le volte che il controllo la "attraversa", e cioe' qualunque sia stato il cammino seguito nella esecuzione del programma per arrivare a tale punto.

Consideriamo il seguente esempio:

```
BEGIN [STAMPA L'IVA DI IMPORTO:]
      TEST PERCENTUALE-IVA
           CASE 2
                TEMP = IMPORTO * 2
           CASE 18
                TEMP = IMPORTO * 18
           CASE 35
                TEMP = IMPORTO * 35
```

```
        ENDTEST ┤TEMP = IMPORTO * PERCENTUALE-IVA├
        TEMP = TEMP / 100
    END ┤TEMP = IVA DI IMPORTO├
```

La seconda asserzione dell'esempio e' corretta soltanto nel caso che la percentuale di IVA non possa assumere altri valori che quelli contemplati dai singoli casi listati.

Particolare attenzione deve essere posta alla scrittura delle asserzioni dentro un ciclo, poiche' queste, per le definizioni poste, devono risultare verificate quando il controllo le "attraversa" ad ogni iterazione.
Esempio:

```
    LEGGI UNA LINEA
    DO UNTIL LINEA LETTA CONTIENE "FINE"
       ┤LETTA UNA LINEA CHE NON CONTIENE "FINE"├
       ELABORA LA LINEA
       LEGGI UNA NUOVA LINEA
    REPEAT
```

1.5 Le asserzioni inserite in un testo procedurale favoriscono la comprensione degli algoritmi.
Consideriamo, ad esempio, il testo seguente:

```
    BEGIN [CALCOLA IN Q E IN R IL VALORE DEL QUOZIENTE
           E DEL RESTO DELLA DIVISIONE DI X PER Y (X ED Y
           INTERI NON NEGATIVI)]
          LEGGI X,Y
          Q = 0
          R = X
          DO WHILE R >= Y
             Q = Q + 1
             R = R - Y
          REPEAT
    END
```

Il funzionamento dell'algoritmo non risulta del tutto evidente dalla sola lettura delle istruzioni: esse non indicano in alcun modo il procedimento logico utilizzato dal programmatore nella loro stesura.

Vediamo ora lo stesso algoritmo con le opportune asserzioni inserite:

```
    BEGIN [CALCOLA IN Q E IN R IL VALORE DEL QUOZIENTE
           E DEL RESTO DELLA DIVISIONE DI X PER Y (X ED Y
           INTERI NON NEGATIVI)]
          LEGGI X,Y
          Q = 0
          R = X
          ┤X = Q * Y + R, CON Q = 0 E R = X├
          DO WHILE R >= Y
             ┤X = Q * Y + R ED R >= Y├
```

```
            Q = Q + 1
            R = R - Y
        REPEAT
        ⊣X = Q * Y + R ED R < Y⊢
    END
```

Il programma sulla base della relazione fondamentale X = Q * Y + R, procede "per tentativi" fino a raggiungere la prima coppia di valori Q ed R per cui e' R < Y.

2. RELAZIONI FRA ASSERZIONI, ASSEGNAMENTI E TEST.

2.0 Abbiamo detto che una asserzione e' una affermazione che deve essere verificata tutte le volte che viene "attraversata" durante l'esecuzione di un programma.
Pertanto, un'asserzione non puo' essere formulata in modo arbitrario, ma deve soddisfare a delle relazioni ben precise sia con le istruzioni del programma che con le altre asserzioni che in esso compaiono.

Si danno tre regole per verificare, senza eseguire il programma, che le asserzioni in esso contenute siano ben formulate.

2.1 REGOLA 1: Se in una sequenza di esecuzione di un programma si incontrano due asserzioni contigue:

```
               .
               .
        ⊣asserzione-1⊢
        ⊣asserzione-2⊢
               .
               .
```

allora deve valere che:

⊣asserzione-1⊢ implica ⊣asserzione-2⊢.

Esempio:

```
     IF ... THEN ...
                 ⊣X > 0⊢
            ELSE ...
                 ⊣X = 0⊢
     ENDIF
     ⊣X >= 0⊢
```

Si hanno nei due cammini:

```
     .                    .
     .                    .
  ⊣X > 0⊢              ⊣X = 0⊢
  ⊣X >= 0⊢             ⊣X >= 0⊢
     .                    .
     .                    .
```

Come si verifica facilmente le asserzioni soddisfano la regola data.

Al contrario, in questo secondo esempio, le asserzioni inserite nel testo non sono ben formulate, come si puo' facilmente verificare:

```
     TEST ...
```

```
          CASE ...
               ...
               ⊣I = 1 E X > 0⊢
          CASE ...
               ...
               ⊣I = 2 E X > 0⊢
          ELSE ...
               ...
               ⊣I = 3⊢
     ENDTEST
     ⊣1 <= I <= 3 E X > 0⊢
```

La regola non e' infatti verificata nel caso venga eseguito il segmento relativo all' ELSE, la proposizione "⊣I = 3⊢" implica ⊣1 <= I <= 3 E X > 0⊢ non e' vera.

2.2 REGOLA 2: Se immediatamente prima della valutazione di una condizione e' verificata l' ⊣asserzione-1⊢, immediatamente dopo la valutazione di tale condizione sara' verificata ⊣⊣asserzione-1⊢ e condizione⊢ oppure ⊣⊣asserzione-1⊢ e non condizione⊢ a seconda che, rispettivamente, la condizione sia vera o falsa.

```
               .
               .
     ⊣asserzione-1⊢
     condizione(vero)
     ⊣⊣asserzione-1⊢ e condizione⊢
               .
               .
```

oppure

```
               .
               .
     ⊣asserzione-1⊢
     condizione(falso)
     ⊣⊣asserzione-1⊢ e non condizione⊢
               .
               .
```

Esempi:

```
     ⊣-10 < X < 10⊢
     IF X >= 0 THEN segmento-1
               ELSE segmento-2
     ENDIF
```

Prima di segmento-1 vale l'asserzione ⊣0 <= X < 10⊢, prima di segmento-2 vale l'asserzione ⊣-10 < X < 0⊢.

```
     ⊣Y > 3⊢
```

```
DO WHILE X > 0
   ...
   ...
   ⊣Y >= 0⊢
REPEAT
```

Dopo la clausola REPEAT vale l'asserzione ⊣X <= 0 e Y >= 0⊢.

2.3 REGOLA 3: Se immediatamente dopo l'istruzione di assegnamento X=espressione e' verificata l' ⊣asserzione-1⊢, allora immediatamente prima della stessa istruzione e' verificata l'asserzione ⊣asserzione-1⊢/X=espressione, cioe' l'asserzione che si ottiene sostituendo in ⊣asserzione-1⊢ ogni occorrenza di X con l'espressione al secondo membro dell'assegnamento.

Tale asserzione si dice che e' stata ottenuta retropropagando ⊣asserzione-1⊢ rispetto all'assegnamento X=espressione.

Esempio.

Se:

```
Z = 0
⊣X = Y E Z = 0⊢
```

con l'applicazione della regola si ottiene:

```
⊣X = Y⊢
Z = 0
⊣X = Y E Z = 0⊢.
```

3. INDICAZIONI PER L'INSERIMENTO DEI COMMENTI.

3.0 Nel caso della sequenza le motivazioni e le asserzioni andranno inserite nel testo procedurale come mostrato nel seguente esempio:

```
BEGIN [motivazione:]
      segmento
      ⊣asserzione⊢
      segmento
      ⊣asserzione⊢
      ........
      ⊣asserzione⊢
      segmento
END
```

3.1 Per la selezione vediamo il caso della struttura TEST-CASE:

```
TEST [motivazione:] espressione
     CASE [motivazione:] lista di casi
          segmento
     .....
     CASE [motivazione:] lista di casi
          segmento
     ELSE ⊣asserzione⊢
          segmento
ENDTEST
```

Ovviamente si suppone che ogni segmento sia gia' opportunamente corredato di motivazioni e asserzioni.

3.2 Per le strutture iterative avremo:

```
DO [motivazione:] UNTIL [motivazione:] condizione
   ⊣asserzione⊢
   segmento
   ⊣asserzione⊢
REPEAT
```

In modo analogo si procedera' per la struttura DO-WHILE-REPEAT.

```
DO [motivazione:]
   ⊣asserzione⊢
   segmento
   ⊣asserzione⊢
REPEAT UNTIL [motivazione:] condizione
```

Analogamente si fara' per la struttura DO-REPEAT-WHILE.

4. CORRETTEZZA.

4.0 L'inserimento di asserzioni nel testo procedurale di un programma contribuisce alla documentazione ed alla leggibilita' del programma stesso da parte dell'uomo, ma non solo. Le asserzioni permettono verifiche di correttezza della sua logica.

Le asserzioni, infatti, rendono esplicite le intenzioni del programmatore all'atto della costruzione di un segmento. Stabiliscono, cioe', le condizioni ritenute valide dal programmatore immediatamente prima e immediatamente dopo ogni esecuzione del segmento in esame.

Dunque, un segmento o un intero programma e' corretto se, supposta vera la sua asserzione di ingresso, le istruzioni in esso contenute sono tali da permettere di dimostrare (nell'ipotesi che la loro esecuzione termini) la verita' dell'asserzione di uscita.

4.1 Uno schema standard di dimostrazione e' il seguente:

si analizza il programma per livelli, applicando, per ogni struttura di controllo, un ben determinato schema di verifica ricavato dalle tre regole elementari precedentemente viste.

4.2 Nel caso della sequenza, per provare che e' corretto:

```
⊣asserzione iniziale⊢
segmento-1
⊣asserzione intermedia⊢
segmento-2
⊣asserzione finale⊢
```

e' sufficiente provare che i due programmi:

```
⊣asserzione iniziale⊢
segmento-1
⊣asserzione intermedia⊢
```

e

```
⊣asserzione intermedia⊢
segmento-2
⊣asserzione finale⊢
```

sono corretti.

4.3 Per provare la correttezza della selezione binaria, per provare cioe' che e' corretto:

```
⊣asserzione iniziale⊢
IF condizione THEN segmento-1
                ELSE segmento-2
ENDIF
⊣asserzione finale⊢
```

e' sufficiente dimostrare che sono corretti i seguenti due programmi:

```
⊣asserzione iniziale e condizione⊢
segmento-1
⊣asserzione finale⊢


⊣asserzione iniziale e non condizione⊢
segmento-2
⊣asserzione finale⊢
```

Se la clausola ELSE manca, invece di dimostrare la correttezza del secondo programma occorrera' dimostrare che:

```
⊣asserzione iniziale e non condizione⊢
implica
⊣asserzione finale⊢
```

Ad esempio sia:

```
⊣0 <= N E X <= 2N⊢
IF X < N THEN X = X + 1
           ELSE X = X - N
ENDIF
⊣X <= N⊢
```

Per provarne la correttezza e' necessario provare che sono corretti:

```
⊣0 <= N E X <= 2N E X < N⊢ X = X + 1 ⊣X <= N⊢


⊣0 <= N E X <= 2N E X < N⊢ X = X - N ⊣X <= N⊢
```

Nella prima, retropropagando ⊣X <= N⊢ rispetto a X = X + 1, si ottiene ⊣X + 1 <= N⊢ o, cio' che e' lo stesso, X < N. Essendo 0 <= N e X <= 2N veri per ipotesi e poiche' X < N implica X < N, la prima e' dimostrata. Analogamente per l'altra.

4.4 Nel caso della selezione per casi, per provare che e' corretto:

```
⊣asserzione iniziale⊢
TEST espressione
     CASE lista-di-casi-1
              segmento-1
```

```
          CASE lista-di-casi-2
                   segmento-2
          ..........
          CASE lista-di-casi-(n-1)
                   segmento-(n-1)
          ELSE
                   segmento-n
     ENDTEST
     ⊣asserzione finale⊢
```

e' sufficiente verificare la correttezza dei seguenti programmi (nell'ipotesi che le liste dei casi siano mutuamente esclusive):

per i = 1,...,n-1

```
     ⊣asserzione iniziale e valore appartenente alla lista-di-casi-i⊢
     segmento-i
     ⊣asserzione finale⊢
```

e

```
     ⊣asserzione iniziale e valore della espressione
      non appartenente a nessuna lista di casi⊢
     segmento-n
     ⊣asserzione finale⊢
```

Nel caso in cui la clausola ELSE non sia specificata, la dimostrazione della correttezza del programma-n va sostituita dalla dimostrazione che:

```
     ⊣asserzione iniziale e valore della espressione
      non appartente a nessuna lista di casi⊢
     implica
     ⊣asserzione finale⊢
```

4.5 Per quanto riguarda la dimostrazione della correttezza nel caso di strutture iterative, consideriamo da prima la struttura DO-REPEAT-UNTIL,

```
     ⊣asserzione iniziale⊢
     DO
        ⊣asserzione-1⊢
        segmento
        ⊣asserzione-2⊢
     REPEAT UNTIL condizione
     ⊣asserzione finale⊢
```

Tale struttura puo' essere considerata equivalente alla seguente, costituita da infiniti IF-THEN-ELSE-ENDIF nidificati:

```
     ⊣asserzione iniziale⊢
```

```
      ⊣asserzione-1⊢
      segmento
      ⊣asserzione-2⊢
      IF condizione THEN nessuna operazione
                    ELSE
                         ⊣asserzione-1⊢
                         segmento
                         ⊣asserzione-2⊢
                         IF condizione THEN nessuna operazione
                                       ELSE ......
                         ENDIF
      ENDIF
      ⊣asserzione finale⊢
```

Pertanto, le regole per ricavare una dimostrazione di correttezza sono le seguenti:

1. verificare che ⊣asserzione iniziale⊢ implica ⊣asserzione-1⊢
2. verificare che ⊣asserzione-2 e condizione⊢ implica ⊣asserzione finale⊢
3. verificare che ⊣asserzione-2 e non condizione⊢ implica ⊣asserzione-1⊢
4. verificare che e' corretto il programma:

```
        ⊣asserzione-1⊢
        segmento
        ⊣asserzione-2⊢.
```

Nel caso in cui ⊣asserzione-1⊢ non sia stata esplicitamente specificata, si puo' assumere che essa sia:

```
        ⊣asserzione-1 e non condizione⊢.
```

Nel caso della struttura DO-UNTIL-REPEAT,

```
        ⊣asserzione iniziale⊢
        DO UNTIL condizione
           ⊣asserzione-1⊢
           segmento
           ⊣asserzione-2⊢
        REPEAT
        ⊣asserzione finale⊢
```

le regole per la dimostrazione della correttezza sono le seguenti:

1. verificare che ⊣asserzione iniziale⊢ implica ⊣asserzione-2⊢
2. verificare che ⊣asserzione-2 e condizione⊢ implica ⊣asserzione finale⊢
3. verificare che ⊣asserzione-2 e non condizione⊢ implica ⊣asserzione-1⊢
4. verificare che e' corretto il programma:

```
        ⊣asserzione-1⊢
        segmento
```

```
        ⊣asserzione-2⊢.
```

Nel caso in cui ⊣asserzione-1⊢ non sia specificata le regole 3. e 4. si riducono a:

3'. verificare che e' corretto il programma:

```
        ⊣asserzione-2 e non condizione⊢
        segmento
        ⊣asserzione-2⊢.
```

Come esercizio si verifichi la correttezza del seguente programma:

```
        BEGIN [CALCOLA IN Z IL PRODOTTO X * Y:]
               LEGGI X,Y
               Z = 0
               U = X
               ⊣Z = 0 E U = X e X > 0⊢
               DO [TOTALIZZA IN Z IL VALORE DI Y:]
                   ⊣X * Y = Z + U * Y E U > 0⊢
                   Z = Z + Y
                   U = U - 1
                   ⊣X * Y = Z + U * Y E U >= 0⊢
               REPEAT UNTIL [TUTTE LE TOTALIZZAZIONI EFFETTUATE:] U = 0
               ⊣Z = X * Y⊢
        END
```

4.6 L'aspetto negativo importante che presenta una dimostrazione formale di correttezza e' la troppa laboriosita' anche nel caso di programmi banali.

Le verifiche di correttezza, pertanto, quando gli algoritmi non richiedano certificazioni particolarmente rigorose, saranno condotte con metodi informali, cioe' il programmatore non produrra' una "dimostrazione" di correttezza, ma tendera' a raggiungere un "ragionevole" grado di convinzione che il programma costruito soddisfi alle specifiche.

Nella scrittura delle asserzioni, non si cerchera' tanto di essere formali ed esaurienti, quanto invece "convincenti".

In conclusione, dunque, il lettore di un programma dovra' essere messo in grado di comprendere, dalla lettura del testo e applicando informalmemte le regole date, come il programma funziona e perche'.

Esempio: programma per il calcolo del prodotto X * Y.

```
      ⊣Z AZZERATO, U = X POSITIVO⊢
      DO [TOTALIZZA IN Z IL VALORE DI Y:]
         ⊣ANCORA U (>0) TOTALIZZAZIONI DA EFFETTUARE⊢
```

```
   Z = Z + Y
   U = U - 1
   ⊣NUOVA TOTALIZZAZIONE EFFETTUATA, U AGGIORNATO⊢
REPEAT UNTIL [TUTTE LE TOTALIZZAZIONI EFFETTUATE:] U = 0
⊣Z = X * Y⊢
```

5.0 In ambienti di produzione di software la documentazione del prodotto ha un ruolo di primaria importanza, non solo ai fini di corredare il prodotto della necessaria descrizione, ma fondamentalmente in quanto e' il principale strumento di comunicazione tecnica che permette di diffondere competenza e cultura e di integrare i progettisti nell'ambiente fisico di lavoro.

I documenti che corredano un programma sono sostanzialmente di tre tipi: le "specifiche funzionali" (cioe' documenti che descrivono le caratteristiche funzionali che un prodotto offre o dovra' offrire), le "specifiche di disegno" (cioe' documenti che descrivono l'architettura generale e la collocazione nell'ambito del sistema), le "specifiche di manutenzione" (documenti che descrivono la struttura interna del prodotto, con un dettaglio sufficiente a permettere interventi sul prodotto anche da parte di chi non abbia partecipato alla sua costruzione).

Nel seguito ci occuperemo di quest'ultimo tipo di documenti, cioe' della documentazione di manutenzione.

5.1 Obiettivo primario dichiarato di questo tipo di documentazione e' quello di permettere una adeguata manutenzione dei prodotti.

Tre aspetti determinano l'esigenza di una buona documentazione di manutenzione: il costo di manutenzione, la "portabilita'" dei programmatori sui vari progetti, la formazione di personale nuovo.

5.2 Il costo di manutenzione e' riconosciuto come il maggiore durante tutta la vita di un programma.

Esso e' determinato dal fatto che gli interventi in un programma ormai stabile sono un fatto che accompagna il programma per tutta la sua vita. Tali interventi sono richiesti generalmente per la domanda di cambiamenti funzionali dovuti a modifiche hardware e/o software dell'ambiente circostante, o per il reperimento di errori durante il suo uso da parte dell'utente esterno.

Gli interventi su un programma in genere si collocano temporalmente molto distanti dal momento del disegno e dell'implementazione.
Cio' crea problemi nella sua conoscenza di dettaglio anche per aspetti di cambiamento di personale, e l'intervento e' tanto piu' costoso quanto meno organizzati e strutturati si presentano i programmi.

Appare dunque fondamentale una buona documentazione che guidi un qualunque progettista alla comprensione di un prodotto che egli non ha implementato.

5.3 Per quanto riguarda la portabilita' dei programmatori sui diversi progetti, va ricordato che in una azienda dove l'implementazione sia distribuita tra numerosi progettisti e' importante poter rapidamente sostituire personale venuto a mancare su un progetto, senza che tale introduzione causi un rallentamento eccessivo per problemi di "messa al passo" del nuovo arrivato.

Un'adeguata mobilita' del personale sui vari progetti permette una migliore distribuzione delle forze di lavoro al variare delle esigenze, evitando di cristallizzare progettisti su progetti specifici anche quando questi siano in una fase di assestamento finale che non richiede un grosso carico di lavoro.

5.4 Rispetto al problema della formazione di nuovo personale, si puo' osservare che di fatto l'acquisizione di competenza tecnica negli ambienti di produzione di software avviene oggi prevalentemente attraverso meccanismi di imitazione dell'ambiente e di perpetuazione di tradizioni che hanno le loro radici nel fatto che il personale neoassunto viene spesso dedicato a manutenzione di programmi.

Pertanto una cattiva documentazione di manutenzione puo' ridurre le possibilita' di crescita del nuovo personale, demotivarlo ed infine costituire un grosso freno a cambiamenti della cultura tecnica di un ambiente di produzione.

5.5 Tradizionalmente nella costruzione della documentazione di manutenzione si incontrano difficolta' che derivano principalmente dal fatto che tale documentazione viene di solito scritta quando ormai il prodotto e' stato completato. La documentazione di manutenzione cosi' non ha piu' alcuna possibilita' di fungere da strumento di riflessione critica che possa influenzare la stesura dei programmi.

Il suo ruolo pertanto viene identificato fondamentalmente solo nel permettere correzioni ai programmi ed e' vissuta dal progettista che la deve scrivere come un compito fastidioso, come una vera e propria fase di lavoro successiva alle precedenti, in cui le sue capacita' tecniche non vengono utilizzate.

Per le ragioni elencate la documentazione di manutenzione e' la prima a perdere di qualita' e quindi di efficacia, in caso di problemi di pianificazione, di ritardi, di esigenza di forza lavoro.

Cio' e' ancora piu' misurabile e generalizzato nel caso di correzioni ai programmi, per cui spesso non viene riportata la corrispondente correzione nella documentazione.

5.6 Sorgono dunque spontanee richieste per uno standard di documentazione e per l'organizzazione della sua costruzione.

E' necessario che la documentazione accompagni la costruzione dei programmi e sia in grado di influenzarla, che sia aggiornata, che permetta un rapido inserimento di persone nuove su un progetto non ancora completo (che cresca quindi in modo sincrono ai programmi), che, infine, possa essere di guida per la formazione di personale di esperienza limitata.

Si puo' soddisfare alle richieste elencate cercando di integrare la documentazione nel programma stesso in modo che la sua lista di compilazione costituisca la parte piu' rilevante della documentazione.
Di conseguenza, la qualita' della documentazione, la mobilita' del personale sui progetti, la crescita del personale saranno funzioni della qualita' del programma.

5.7 Una tale documentazione dovra' "guidare" il lettore alla comprensione di programmi anche di rilevanti dimensioni, per gradi, con un rispetto della propedeuticita' della presentazione di nuove informazioni.

Sara' necessario che la documentazione (il programma!) sia suddivisa in moduli organizzati in modo gerarchico, il piu' elevato dei quali descriva l'intero programma in termini di moduli di livello inferiore, e cosi' via, fino ai livelli elementari.

Si richiede, cioe', l'adozione di tecniche di programmazione strutturata top-down.

Gli strumenti linguistici impiegati nella costruzione della lista di compilazione autodocumentata non potranno che essere lo stesso linguaggio di programmazione, accompagnato da un uso adeguato di commenti.

Uno standard di organizzazione del programma, dell'uso di commenti e quindi dell'organizzazione del listato, sara' in grado di determinare in modo completo la costruzione di programmi autodocumentati, rendendo necessariamente molto aderente la documentazione del programma alla sua reale struttura.

6.0 Per la buona descrizione della struttura e delle caratteristiche di dettaglio di un programma e' necessario fornire:

- una testata che contenga le "specifiche di targa" per individuare il prodotto nell'ambito del sistema;

- indicazioni sulla struttura globale del prodotto in termini della sua organizzazione gerarchica (tali indicazioni possono essere direttamente fornite sulla lista del programma mediante la descrizione dell'albero che rappresenta la gerarchia del programma, effettuata mediante commenti);

- la descrizione dei dati, in particolare di ingresso e uscita e le aree correlate, effettuata direttamente mediante le istruzioni del linguaggio di programmazione, opportunamente commentate;

- la descrizione delle diverse porzioni del programma, di diverso livello gerarchico, inquadrando, per ciascuna porzione, le sue caratteristiche dal punto funzionale e le sue caratteristiche dal punto di vista fisico, in particolare in relazione agli aspetti che maggiormente interessano interfacce con altri moduli.

6.1 La descrizione dei moduli deve contenere:

1. il nome del modulo;

2. una descrizione della trasformazione che il modulo effettua, da un punto di vista puramente funzionale, con la specificazione degli obiettivi che tale trasformazione si propone di raggiungere;

3. le condizioni di ingresso, cioe' la descrizione dei dati in ingresso al modulo dal punto di vista del loro significato in termini di problema (che entita' o informazioni essi rappresentano, e la descrizione delle aree di memoria corrispondenti ai dati in ingresso, dei relativi valori possibili o reali, con riferimento ai moduli da cui provengono i dati;

4. una descrizione dell'algoritmo impiegato per effettuare la trasformazione, eventualmente coinvolgente i nomi dei dati descritti;

5. le condizioni di uscita, in modo analogo alle condizioni di ingresso;

6. le aree di lavoro, cioe' una descrizione di tutte quelle aree che non essendo ne' di ingresso ne' di uscita, vengono usate all'interno del modulo: spesso in esse si trovano le interfacce con i moduli richiamati;

7. l'elenco dei moduli richiamati da quello in esame, eventualmente corredato da una brevissima descrizione della funzione svolta da ogni modulo;

8. le interfacce dati con tali moduli, cioe' l'elenco delle aree interessate nella comunicazione tra i moduli, corredato eventualmente di valori possibili e del relativo significato;

9. i trasferimenti di controllo ("da" e "a") non organizzati in modo gerarchico (ad esempio, passaggi da una sezione del programma ad una successiva);

10. note aggiuntive che il programmatore ritiene utili alla comprensione del programma.

Per ciascun modulo, alla parte di listato descritta deve seguire il codice che realizza il modulo stesso, di dimensioni tali da raggiungere, con la parte di descrizione illustrata, la dimensione media di due pagine di listato, al fine di mantenere snello il lavoro di lettura e consultazione.

Il metodo di autodocumentazione prevede l'inserimento guidato di commenti intermedi inseriti nel codice stesso, classificati secondo le due fondamentali categorie: motivazioni e asserzioni.

7. SVILUPPO TOP-DOWN

7.0 Un programmatore, in generale, non avra' il solo compito di descrivere in modo ordinato delle sequenze di esecuzione a lui note. Dovra' svolgere un'attivita' di duplice natura: posto di fronte a un problema da risolvere con un programma, da un lato dovra' preconfigurare il processo di azioni con cui tale problema puo' essere risolto, e dall'altro dovra' descrivere tale processo in modo ordinato, utilizzando gli strumenti linguistici opportuni.

Gli aspetti di "invenzione" sono intrinseci all'attivita' di programmazione. E' possibile pero' dare un insieme di linee guida che e' conveniente da adottare nella costruzione di un algoritmo, che permettano una rapida verifica, ad ogni passo successivo del processo di invenzione, della ragionevolezza delle scelte effettuate.

7.1 L'idea base del procedimento di stesura per raffinamenti successivi (o top-down) e' assai semplice.
Quando la complessita' del problema da risolvere cresce, non diventa piu' possibile tenere conto contemporaneamente di tutti gli aspetti coinvolti, fino al massimo livello di dettaglio, e prendere contemporaneamente tutte le decisioni realizzative. In tale caso sara' necessario procedere "secondo approssimazioni successive", cioe' decomporre il problema iniziale in sottoproblemi piu' semplici e sviluppare il programma immaginando che tali sottoproblemi vengano risolti da altri programmi di livello gerarchico inferiore.
Si affrontera' quindi ciascuno dei sottoproblemi in modo analogo.

Le tecniche di sviluppo top-down suggeriscono di scrivere subito il programma completo, come se il linguaggio di programmazione a disposizione fosse di livello molto elevato e orientato al problema in esame.
Tale programma conterra', oltre alle solite strutture di controllo, anche istruzioni di alto livello, cioe' nomi di operazioni complesse che dovranno poi essere ulteriormente specificate. Queste operazioni verranno poi descritte in termini di operazioni ancora piu' semplici, e cosi' via fino ad arrivare alle operazioni elementari fornite dal linguaggio di programmazione utilizzato.

Ad ogni passo successivo dello sviluppo si procedera' inoltre a una verifica (informale) della correttezza del raffinamento appena effettuato.

8. CONCLUSIONI E RACCOMANDAZIONI.

8.0 Abbiamo detto che un programma non e' solo uno strumento di comunicazione con il calcolatore che lo deve eseguire: deve essere anche uno strumento di comunicazione per l'uomo.

Abbiamo osservato anche che raramente un programma esaurisce la sua vita nell'arco di poche esecuzioni: dovra' in genere sopravvivere a modifiche dell'ambiente in cui esso e' stato costruito, dovute sia a cambiamenti fisici dell'hardware, sia a cambiamenti del software di sistema. Un programma, dunque, potra' richiedere svariate modifiche, nel corso della sua vita, per far fronte a nuove esigenze.

Abbiamo pertanto sottolineato l'importanza che il programma sia facilmente comprensibile non solo al programmatore che l'ha scritto, anche a distanza di molti mesi, ma anche ad altri programmatori che ne possano prendere il posto, i quali devono essere in grado di effettuare le modificazioni necessarie in modo completamente autonomo.

In genere e' rivolta a questi obiettivi la documentazione di disegno del programma, cioe' quell'insieme di documenti che descrivono le caratteristiche dell'algoritmo scelto, lo scopo delle diverse aree di memoria, il modo con cui il programma e' suddiviso, ecc.

8.1 La documentazione generalmente, e' stato osservato, e' un prodotto posteriore alla stesura del programma, e non e' pertanto di alcuna utilita' nella sua costruzione, mentre non fa che aumentare il costo complessivo dello sviluppo. Inoltre, spesso essa riflette una visione troppo personale degli aspetti importanti da descrivere, cosicche' non e' di grande utilita' a chi non l'ha scritta.
Infine, una volta che il prodotto viene modificato, raramente la documentazione viene aggiornata, oppure viene aggiornata in modo incompleto.

Buon metodo sara' al contrario calare nello stesso codice del programma, per quanto possibile, la documentazione stessa, cioe' costruire programmi facilmente leggibili ed autodocumentantisi.

Un programma che si autodocumenta e che risulta di facile comprensione anche a chi non l'abbia scritto non solo evita la stesura di pesanti documenti, ma fa si' che la documentazione viva con il programma, cambi e cresca con esso, e quindi sia sempre aggiornata.
Inoltre essa viene necessariamente scritta nella fase stessa di sviluppo, e risulta uno strumento molto utile per prendere decisioni sulla struttura dell'algoritmo.

8.2 Come si e' visto, strumenti necessari per favorire la leggibilita' e l'autodocumentazione sono le strutture di controllo, le

motivazioni, le asserzioni, ma essi non sono sufficienti.

Per guidare ad una lettura per livelli e' utile la segmentazione, cioe' mantenere anche nel programma definitivo fisicamente separate le porzioni di codice che si riferiscono ai diversi livelli.
Si tratta cioe' di considerare programmi completi ed eseguibili non solo il programma finale scritto nel linguaggio di programmazione, ma anche le versioni intermedie, come ad esempio:

```
BEGIN
     DO VARYING K FROM 2 TO 100
             INSERISCI K-ESIMO PRIMO IN TABELLA
     REPEAT
END
```

in cui compaiono "istruzioni ad alto livello", la cui specificazione in termini di codice eseguibile puo' essere fatta a parte.

Per facilitare l'individuazione delle istruzioni ad alto livello del tipo indicato si introduce il verbo PERFORM; ad esempio:

```
PERFORM INSERISCI K-ESIMO PRIMO IN TABELLA
```

Per definire (a parte) tali istruzioni con codice di livello piu' basso, useremo le parole chiave PROC e ENDPROC:

```
PROC INSERISCI K-ESIMO PRIMO IN TABELLA
    ..........
    codice che realizza l'"istruzione"
    ..........
ENDPROC
```

8.3 L'uso della PERFORM tuttavia non risolve tutte le situazioni in cui noi siamo interessati ad una suddivisione fisica del codice del programma.
Esiste anche il caso dei predicati, cioe' di quelle porzioni di programma il cui scopo e' quello di verificare l'occorrenza di una certa condizione, fornendo un risultato di tipo "vero" o "falso".

E' preferibile esprimere direttamente nel codice una frase del tipo:

```
REPEAT UNTIL PRIMO-J
```

specificando a parte come PRIMO-J debba essere calcolato, piuttosto che oscurare la porzione di codice relativa con una lunga serie di istruzioni destinata alla verifica che J sia primo o meno. Per comprendere il livello in cui e' presente la clausola REPEAT UNTIL PRIMO-J, infatti, non abbiamo bisogno di conoscere in dettaglio come si debba valutare PRIMO-J.

Anche in questo caso, converremo di specificare a parte tali predicati, in una porzione di codice individuata attraverso le parole

chiave PRED e ENDPRED:

```
PRED PRIMO-J
     codice che controlla la condizione
     e restituisce un valore VERO o FALSO
ENDPRED
```

8.4 Il testo definitivo del programma deve permettere al lettore di occuparsi di poche cose er volta, eventualmente anche in un ordine diverso da quello con cui le varie porzioni di codice sono state pensate o scritte.

L'uso delle PERFORM fornisce un programma di livello alto che e' il riassunto dell'intero programma, e permette una lettura del codice per livelli di dettaglio successivi.

Tuttavia, risulta molto utile anche la possibilita' di comprendere a fondo le funzioni svolte da un solo componente, senza doversi occupare dei dettagli degli altri componenti dello stesso livello o dei livelli superiori.
A tale scopo si dovra' cercare di considerare ogni componente come un vero e proprio programma completo, inserito in una gerarchia di programmi che si richiamano tra loro, destinato ad effettuare una precisa trasformazione, vincolato ad accettare determinate condizioni di ingresso e determinate condizioni di uscita.

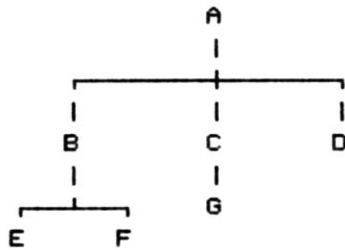
8.5 Ciascun livello va scritto in modo che possa esser letto in modo completamente autonomo e staccato dal resto, proprio come se si trattasse di un programma a se' stante.

Pertanto sono da adottare opportune tecniche di impaginazione in modo che i diversi livelli siano ben distinguibili: ciascun livello sara' ampiamente commentato, separato dagli altri da file di asterischi o di trattini, e le sue dimensioni saranno quelle di una o due pagine al massimo, allo scopo di poterlo tenere sott'occhio tutto in una sola volta.
Inoltre si fara' precedere il codice di ciascuna porzione da una descrizione della trasformazione che esso effettua e da un'asserzione che specifichi quali sono le condizioni che richiede siano presenti in ingresso; e si fara' seguire il codice da un'asserzione che specifichi qual e' lo stato raggiunto dopo l'esecuzione.

8.6 Un ultimo problema che si presenta e' quello di indicare in modo rapido e schematico come tutte le diverse porzioni di codice, che possono essere disordinate rispetto alla struttura del programma, siano correlate tra loro e quali siano i rapporti di dipendenza gerarchica con cui nel programma esse vengono richiamate.

A tale scopo si introduce una descrizione sintetica del programma, una specie di "indice", che permetta di vedere rapidamente la struttura.

Tale descrizione consiste nell'albero dei richiami, che illustra graficamente la struttura gerarchica dell'intero programma:



L'albero puo' essere letto per "livelli orizzontali" o per "scansioni verticali".